Engineering High-Performance Community Detection Heuristics for Massive Graphs

Christian L. Staudt and Henning Meyerhenke
Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Germany
{christian.staudt, meyerhenke}@kit.edu

***Abstract*—The amount of graph-structured data has recently experienced an enormous growth in many applications. To transform such data into useful information, high-performance analytics algorithms and software tools are necessary. One common graph analytics kernel is community detection (or graph clustering). Despite extensive research on heuristic solvers for this task, only few parallel codes exist, although parallelism is often necessary to scale to the data volume of real-world applications.**

We address the deficit in computing capability by a flexible and extensible clustering algorithm framework with shared-memory parallelism. Within this framework we implement our parallel variations of known sequential algorithms and combine them by an ensemble approach. In extensive experiments driven by the algorithm engineering paradigm, we identify the most successful parameters and combinations of these algorithms. The processing rate of our fastest algorithm exceeds 10M edges/second for many large graphs, making it suitable for massive data streams. Moreover, the strongest algorithm we developed yields the best tradeoff between quality and speed for graph clusterers to date.

Keywords: Community detection, graph clustering, high-performance network analysis, parallel algorithm engineering

I. Introduction

The data volume produced by electronic devices is growing at an enourmous rate. An important class of such data is structured as or can be modeled by graphs. For instance, online social networks are increasingly popular, the largest being Facebook with more than 600 million daily active users.[1] The WWW forms a network of hyperlinked webpages in excess of 30 billion nodes. To be able to analyze network data of this magnitude in near real-time, algorithms and hardware have to keep up with these data volumes and rates. However, only few algorithms are able to handle such massive inputs. A particular challenge is not only the size of the data, but also its structure. *Complex networks*, in contrast to regular meshes, have topological features which pose computational challenges: In a *scale-free* network, the presence of a few high-degree nodes (hubs) generates load balancing issues. In a *small world* network, the entire graph can be visited in only a few hops from any source node, which negatively affects cache-performance. This increases the importance of structural network analysis. As a result, costly algorithms can be applied only to certain relevant parts of the network after they have been identified in a precedent analysis.

[1]Facebook key facts: http://newsroom.fb.com/Key-Facts

In this work, we approach the task of *community detection* in networks, also known as *graph clustering*, with a focus on scalability. Applications are manifold, from counteracting search engine rank manipulation [18] to discovering scientific communities in publication databases [20]. So far, extensive research on graph clustering has given rise to a variety of definitions of what constitutes a good community and a variety of methods for finding such communities, many of which are described in surveys by Schaeffer [18] and Fortunato [6]. Among these definitions, the lowest common denominator is that a community or cluster is an internally dense node set with sparse connections to the rest of the graph. The clustering quality measure *modularity [9]* formalizes the notion of a good clustering by comparing its *coverage* (fraction of edges within a cluster) to an expected value based on a random edge distribution model. Modularity is not without flaws [7] nor alternatives [21], but has emerged as a well-accepted measure of clustering quality. This makes modularity our measure of choice for evaluating our results. While optimizing modularity is NP-hard [3], efficient heuristics have been introduced which explicitly increase modularity: A globally greedy agglomerative method [4] runs in $O(md \log n)$ for graphs with n nodes and m edges, where d is the depth of the dendrogram of mergers and typically $d \sim \log n$. A locally greedy multilevel-algorithm known as the *Louvain method* [2] has been experimentally shown to be three orders of magnitude faster than Clauset et al.'s [4] agglomeration. Noack and Rotta [17] present another multilevel algorithm, which combines agglomeration with refinement.

A. Motivation

For graphs with millions or billions of edges, only (near) linear-time clustering algorithms can be considered in practice. Several fast clustering methods have been developed in recent years. There is, however, a lack of research in adapting these methods to take advantage of parallelism. A very recent attempt at assessing the state of the art in community detection was conducted by the *10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering* [1]. *DIMACS* challenges are scientific competitions in which the participants solve problems from a specified test set, with the aim of high solution quality and high execution performance. Ten solver families were submitted (with a total of 15 different

implementations) for optimizing modularity. However, only two of them relied on parallel processing and only very few could handle the largest graphs with billions of edges in a reasonable amount of time.

Accordingly, our objective is the development and implementation of a highly parallel clustering heuristic which is able to handle massive graphs quickly while also producing a high-quality clustering. In the following, the data sets and results from the *DIMACS* challenge will serve as a major benchmark for our own work presented here.

B. Methods

We implement two parallel algorithms, each of which can be used as a relatively efficient standalone clusterer. In addition, we also implement a two-phase approach that combines their strengths: The problem size is first reduced in a preprocessing phase focusing on speed. Afterwards, a more expensive method is applied which emphasizes quality maximization. The preprocessing phase is inspired by a machine learning strategy known as *ensemble learning* [14], in which the output of several weak classifiers is combined to form a strong one. In our case, multiple *base clusterers* run in parallel as an ensemble. Their clusterings are then combined to form a *core clustering*, representing the consensus of all base clusterers. The graph is contracted according to the core clustering, and then assigned to a single *final clusterer*. Finally, the resulting clustering of the coarse graph is applied to the input graph. Within this extensible framework, which we call the *ensemble preprocessing* method (EPP), we apply a parallel *label propagation* algorithm (PLP) as base clusterers and a parallel variant of the *Louvain method* (PLM) as the final clusterer. Label propagation is a very simple procedure where nodes adopt the cluster assignment (label) which is most frequent among their neighbors until a stable clustering emerges. The Louvain method is a multilevel technique in which nodes are repeatedly moved to the clusters of a neighbor. We combine multiple PLP base clusterings with a highly parallel hashing-based scheme, implicitly finding nodes at the boundary of communities whose affiliation is disputable. Afterwards we considerably reduce the problem size by graph contraction, and then allow the slower but qualitatively superior PLM to maximize modularity, thus investing extra time into classifying boundary nodes.

C. Capabilities

With our shared-memory parallel implementation of label propagation clustering (PLP), we provide an extremely fast basic clustering algorithm that scales well with the number of processors. The processing rate of PLP exceeds 10^7 edges per second for many large graphs, making it suitable for massive data streams (see Figure 1, and Section VI-A for a description of the graphs). With PLM, we present the first parallel implementation of the Louvain clustering method for massive inputs. The EPP ensemble algorithm combines

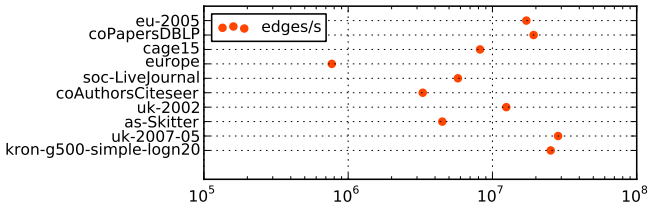

Figure 1. Processing rate in edges per second of the PLP algorithm on large graphs.

the advantages of PLP and PLM. After the fast PLP preprocessing phase, even the largest graphs become tractable for the qualitatively strong PLM. To our knowledge, EPP yields the best tradeoff between quality and speed for graph clusterers to date. Our implementation outperforms state-of-the-art algorithms with respect to either speed or clustering quality, as evidenced by comparison to the *DIMACS Challenge* results (see Section VII-D).

Our clustering algorithm framework is flexible and extensible, supporting rapid iteration between design, implementation and testing required for algorithm engineering [12]. In this work, we focus on specific configurations of clustering algorithms, but novel combinations can be quickly evaluated in the future.

II. Related Work

This section gives a short overview over related efforts. For a comprehensive overview of graph clustering, we refer the interested reader to the aforementioned surveys [18], [6]. Recent developments and results are covered by the *10th DIMACS Implementation Challenge* [1].

Clustering by label propagation has originally been described by Raghavan et al. [15], and several variants of the algorithm exist. One of these variants under the name *peer pressure clustering* is due to Gilbert et al. [8], who use the algorithm as a prototype application within a parallel toolbox that uses numerical algorithms for combinatorial problems. Unfortunately, the latter report running times only for a different clustering algorithm, which solves a very specific benchmark problem and is not applicable in our context.

Ovelgönne and Geyer-Schulz [14] apply the *ensemble learning* paradigm to graph clustering. They develop what they call the *Core Groups Graph Clusterer* (CGGC) scheme, which we adapt and evaluate as the *Ensemble Preprocessing* (EPP) algorithm. They also introduce an iterated scheme (CGGCi) in which the core clustering is again assigned to an ensemble, creating a hierarchy of clusterings/contracted graphs until clustering quality does not improve any more. We also implement a variation of the iterated scheme as the *Ensemble Multilevel* (EML) algorithm. Within this framework, they test two base clusterers: *Randomized Greedy* (RG), a variant of the aforementioned algorithm by Clauset et al. [4], avoids a loss in clustering quality that arises from highly unbalanced cluster sizes. Secondly, a sequential implementation of the

label propagation clustering scheme is used. CGGC with RG as base and final clusterers competes in the *DIMACS* challenge under the name RG+, and emerged as the winner of the Pareto part of the challenge, which related quality to speed in order to elect the best tradeoff.

Recently Ovelgönne [13] presented a distributed implementation (based on the big data framework *Hadoop*) of an ensemble preprocessing scheme using label propagation as a base algorithm. This implementation processes a 3.3 billion edge web graph in a few hours on a 50 machine Hadoop cluster [13, p. 10]. Our *OpenMP*-based implementation of the similar EPP algorithm requires only a few *minutes* on one shared-memory machine with 16 physical cores.

Among the few parallel clusterers competing in the *DIMACS* challenge, Fagginger Auer and Bisseling [5] submitted an agglomerative clusterer with an implementation for both the GPU (using *NVIDIA CUDA*) and the CPU (using *Intel TBB*). The algorithm weights all edges with the difference in modularity resulting from a contraction of the edge, then computes a heavy matching M and contracts according to M. This process continues recursively with a hierarchy of successively smaller graphs. The matching procedure can adapt to star-like structures in the graph to avoid insufficient parallelism due to small matchings. In the challenge, the CPU implementation competed as CLU_TBB and proved exceptionally fast. Independently, Riedy et al. [16] developed a similar clusterer, which follows the same principle but does not provide the adaptation to star-like structures.

Other parallel efforts include an algorithm by Zhang et al. [22], which identifies communities based on a custom metric rather than modularity. More closely related to our work is a variant of label propagation by Soman and Narang [19] for multicore and GPU architectures, which seeks to improve quality by re-weighting the graph, and has been shown to cluster a graph with about 100 million edges in a few minutes on an *IBM Power6* system.

III. Preliminaries

We denote a graph, the abstraction of a network data set, as $G = (V, E)$ with a node (or, interchangeably, vertex) set $V$of size n and an edge set E of size m. In the following, edges $\{u, v\}$ are undirected and have weights $\omega : E \to \mathbb{R}^+$. The weight of a set of nodes is denoted as $\omega(E') := \sum_{\{u,v\} \in E'} \omega(u, v)$. A clustering $\zeta = \{C_1, \ldots, C_k\}$ is a partition of the node set V into disjoint subsets called clusters. Equivalently, a clustering can be understood as a mapping $\zeta : V \to \mathcal{P}(V)$ where $\zeta(v)$ returns the cluster containing node v. For our implementation, the nodes have consecutive integer identifiers $id(v)$ in the range $[0, n-1]$ and edges are pairs of node identifiers. A clustering is represented as an array of size n, indexed by integer node identifiers and containing integer cluster identifiers, i.e. a mapping $\zeta : \mathbb{N} \to \mathbb{N}$.

IV. Algorithms

In this section we formulate and describe our parallel variants of existing sequential clustering algorithms, as well as ensemble techniques which combine them. Implementation details are discussed in Section V.

A. Parallel Label Propagation

Label propagation clustering, as originally introduced by Raghavan et al. [15], extracts a clustering from a labelling $V \to \mathbb{N}$ of the node set. Initially, each node is assigned a unique label, and then multiple iterations over the node set are performed: In each iteration, every node adopts the most frequent label in its neighborhood (breaking ties arbitrarily and uniformly). Densely connected groups of nodes thus agree on a common label, and eventually a globally stable consensus is reached, which usually corresponds to a good clustering of the network. Label propagation therefore finds a clustering in near linear time: Each iteration takes $O(m)$ time, and the algorithm has been empirically shown to reach a stable solution in only a few iterations, though not mathematically proven to do so. The number of iterations seems to depend more on the graph structure than the size. More theoretical analysis is done by Kothapalli et al. [11]. The algorithm performs updates to an array of n labels and does not involve the computation of an objective function. Thus, it maximizes modularity (or similar measures) only implicitly. Due to its local update rule, label propagation is well suited for a fast and parallel implementation.

Algorithm 1 denotes PLP, our parallel variant of label propagation. We adapt the algorithm in a straightforward way to make it applicable to weighted graphs. Instead of the most frequent label, the *heaviest label* in the neighborhood is chosen, i.e. the label l that maximizes $\sum_{u \in N(v): \zeta(u) = l} \omega(v, u)$. Iteration continues until the number of nodes which changed their labels falls below a threshold θ. We implement a number of modifications to the original algorithm, described in more detail in Section V-A.

B. Parallel Louvain Method

The algorithm known as the *Louvain method* was first presented by Blondel et al. [2]. It can be classified as a locally greedy agglomerative (bottom-up) algorithm with modularity as an objective function. In each pass, nodes are repeatedly moved to neighboring clusters so that the locally maximal increase in modularity is achieved, until the clustering is stable. Then, the graph is contracted according to the clustering and the procedure continues recursively, forming clusters of clusters. Finally, the clustering of the coarsest graph determines the clustering of the input graph by direct prolongation.

Note that the change in modularity resulting from a node move can be calculated by scanning only the local neighborhood of a node: Let $\omega(u, C) := \sum_{\{u,v\}: v \in C} \omega(u, v)$ be the weight of all edges from u to nodes in cluster C, and define the *volume* of a node and a cluster as

Algorithm 1: PLP: Parallel Label Propagation

Input: graph $G = (V, E)$
Result: clustering $\zeta : V \to \mathbb{N}$

```
parallel for v ∈ V
    ζ(v) ← id(v)
updated ← n
V_active ← V
while updated > θ do
    updated ← 0
    parallel for v ∈ {u ∈ V_active : deg(u) > 0}
        l* ← arg max_l { Σ_{u∈N(v):ζ(u)=l} ω(v,u) }
        if ζ(v) ≠ l* then
            ζ(v) ← l*
            updated ← updated + 1
            V_active ← V_active ∪ N(v)
        else
            V_active ← V_active \ {v}
return ζ
```

Algorithm 2: PLM: Parallel Louvain Method

Input: graph $G = (V, E)$
Result: clustering $\zeta : V \to \mathbb{N}$

```
ζ ← ζ_singleton(G)
anychange ← false
repeat
    done ← true
    parallel for u ∈ V
        δ ← max_{v∈N(u)} {Δmod(v, ζ(u) → ζ(v))}
        C ← ζ(arg max_{v∈N(u)} {Δmod(v, ζ(u) → ζ(v))})
        if δ > 0 then
            ζ(u) ← C
            done ← false
            anychange ← true
until done
if anychange then
    G' ← contract(G, ζ)
    ζ ← prolong(PLM(G'))
return ζ
```

Algorithm 3: EPP: *Ensemble Preprocessing*

Input: graph $G = (V, E)$
Result: clustering $\zeta : V \to \mathbb{N}$

```
parallel for base ∈ B
    ζ_i ← base_i(G)
ζ̄ ← combine(ζ_1, ..., ζ_b)
G^1 ← contract(G, ζ̄)
ζ^1 ← final(G^1)
ζ ← project(ζ^1, G)
return ζ
```

$vol(u) := \sum_{\{u,v\}:v\in N(u)} \omega(u,v) + 2 \cdot \omega(u,u)$ and $vol(C) := \sum_{u\in C} vol(u)$, respectively. The difference in modularity when moving node $u \in C$ to cluster D is then:

$$\Delta mod(u,\ C \to D) = \frac{\omega(u, D \setminus \{u\}) - \omega(u, C \setminus \{u\})}{\omega(E)} + \frac{(vol(C \setminus \{u\}) - vol(D \setminus \{u\})) \cdot vol(u)}{2 \cdot \omega(E)^2}$$

We introduce a shared-memory parallelization of the Louvain method (PLM, Algorithm 2) in which node moves are evaluated and performed in parallel instead of sequentially. This approach may work on stale data so that a monotonous modularity increase is no longer guaranteed. Suppose that during the evaluation of a possible move of node u other threads might have performed moves that affect the Δmod scores of u. In some cases this can lead to a move of u that actually decreases modularity. Still, such undesirable decisions can also be corrected in a following iteration, which is why the solution quality is not necessarily worse. Working only on independent sets of vertices in parallel does not provide a solution since the sets would have to be very small, limiting parallelism and/or leading to the undesirable effect of a very deep coarsening hierarchy. Concerns about termination turned out to be theoretical for our set of benchmark graphs, all of which can be successfully clustered with PLM. We describe implementation aspects in Section V-B and discuss the difference between sequential and parallel solutions in Section VII-B.

C. Ensemble Techniques

In the area of machine learning, *ensemble learning* is a strategy in which multiple *base classifiers* or *weak classifiers* are combined to form a strong classifier. It has been shown that combining results of classifiers only slightly better than random guessing yields qualitatively good results. Classification in this context can be understood as deciding whether a pair of nodes should belong to the same cluster. We follow this general idea, which has been applied successfully to graph clustering before [14]. Subsequently, we describe two ensemble techniques EPP and EML of which EPP emerges as a highly efficient clusterer. We also quickly describe algorithms for combining multiple base clusterings.

1) Ensemble Preprocessing: When aiming for a good trade-off between speed and quality, the following approach emerged as the most promising one: In a preprocessing step, assign G to an ensemble of base clusterers. The graph is then contracted according to the *core clustering* $\bar{\zeta}$, which represents the consensus of the base clusterers (see Section IV-C3 for details). This contraction reduces the problem size considerably, and implicitly identifies the contested and the unambiguous parts of the graph. After the preprocessing phase, the contracted graph G' is assigned to the final clusterer, whose result is applied to the input graph by prolongation. We instantiate this scheme with PLP as a base clusterer and PLM as the final clusterer. Thus we achieve massive nested parallelism with several parallel PLP instances running concurrently in the first phase, and proceed in the second phase with the more expensive but qualitatively superior PLM. This constitutes the

EPP algorithm (Algorithm 3). We write EPP_b to indicate the number of PLP base clusterers.

2) Ensemble Multilevel: A natural way to extend the ensemble preprocessing method is to apply it recursively: After the core clustering has been computed, the original graph G is contracted to a smaller graph G' according to the clustering. Then the algorithm is called recursively on G', again assigning the contracted graph to an ensemble. Several options for stopping the recursion are possible: (i) If the clustering remains the singleton clustering and thus $G = G'$, (ii) if the coarsest graph is smaller than a threshold, (iii) if the change in modularity from one recursion level to the next is non-positive, or (iv) if the quality of the clustering has not improved for a number of levels. Clearly, option (i) requires the program to be stopped to prevent an infinite loop. The other options are not strictly necessary, but save running time in case further quality improvements are unlikely.

3) Core Clustering and Graph Contraction: A consensus of $b > 1$ base clusterers is formed by combining the base clusterings ζ_i in the following way: Only if a pair of nodes is classified as belonging to the same cluster in every ζ_i, then it is assigned to the same cluster in the core clustering $\bar{\zeta}$. Formally, for all node pairs $u, v \in V$:

$$\forall i \in [1, b] : \ \zeta_i(u) = \zeta_i(v) \quad \Longleftrightarrow \quad \bar{\zeta}(u) = \bar{\zeta}(v). \quad \text{(IV.1)}$$

We introduce a combination algorithm based on *hashing*. With a suitable hash function $h(\zeta_1(v), \ldots, \zeta_b(v))$, the cluster identifiers of the base clusterings are mapped to a new identifier $\bar{\zeta}(v)$ in the core clustering. Except for unlikely hash collisions, a pair of nodes will be assigned to the same cluster only if the criterion above is satisfied.

If base clusterings with connected clusters need to be converted to a core clustering with all clusters connected, an alternative approach may be necessary. We suggest a method inspired by *region growing*. Starting with a singleton clustering $\bar{\zeta}$, every edge $\{u, v\}$ is traversed in a breadth-first search of the graph and nodes are assigned according to the rule

$$\forall \zeta_i : \zeta_i(u) = \zeta_i(v) \quad \Longrightarrow \quad \bar{\zeta}(v) \leftarrow \bar{\zeta}(u).$$

However, this approach is relatively slow, does not easily support parallelism, and may not yield clusters according to Eq. (IV.1).

Graph contraction according to a clustering is performed in a straightforward way such that the nodes of a cluster in G are aggregated to a single node in G'. An edge between two nodes in G' receives as weight the sum of weights of inter-cluster edges in G, while self-loops preserve the weight of intra-cluster edges.

V. Implementation

The language of choice for all implementations is *C++11*, allowing us to use object-oriented and functional programming concepts while also compiling to native code. We implemented all algorithms on top of a simple custom adjacency array graph data structure. A high-level interface encapsulates the data structure and enables a clear and concise notation of graph algorithms. In particular, our interface conveniently supports parallel programming through parallel node and edge iteration methods which receive a function (generally a closure) and apply it to all elements in parallel. Parallelism is achieved in the form of loop parallelization with *OpenMP*, using the `parallel for` directive with `schedule(guided)` where appropriate for improved load balacing.

We publish our source under a permissive free software license to encourage reproduction, reuse and contribution by the community.[2]

What follows are details on the implementation of algorithms from Section IV.

A. Parallel Label Propagation

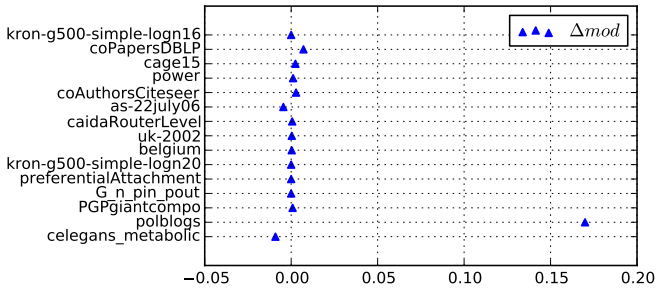

Figure 2. In general, only small modularity differences result from an explicit randomization of PLP node order

In the original description [15], nodes are traversed in random order. Since the cost of explicitly randomizing the node order in parallel is not insignificant, we make this optional and rely on some randomization through parallelism otherwise. We also observe that forgoing randomization has a negligible effect on quality for nearly all graphs (Figure 2).

We avoid unneccessary computation by distinguishing between active and inactive nodes. It is unneccessary to recompute the label weights for a node whose neighborhood has not changed in the previous iteration. Nodes which already have the heaviest label become inactive, and are only reactivated if a neighboring node is updated. We restrict iteration to the set of active nodes. Iterations are repeated until the number of nodes updated falls below a threshold value. The motivation for setting threshold values other than zero is that on some graph instances, the majority of iterations are spent on updating only a very small fraction of high-degree nodes (see Figure 3). Since preliminary experiments have shown that time can be saved and clustering quality is not significantly degraded by simply omitting these iterations, we set an update threshold of $\theta = n \cdot 10^{-5}$. Note that we do not use the termination criterion specified in [14] as it does not lead to convergence on some inputs. In the original description [15], the criterion is to stop

[2]open-source release: http://parco.iti.kit.edu/software/

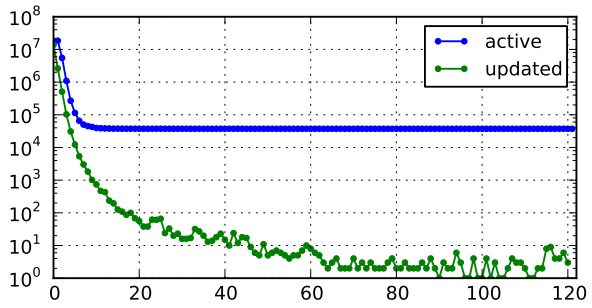

Figure 3. Number of active and updated labels per iteration of PLP for graph uk-2002.

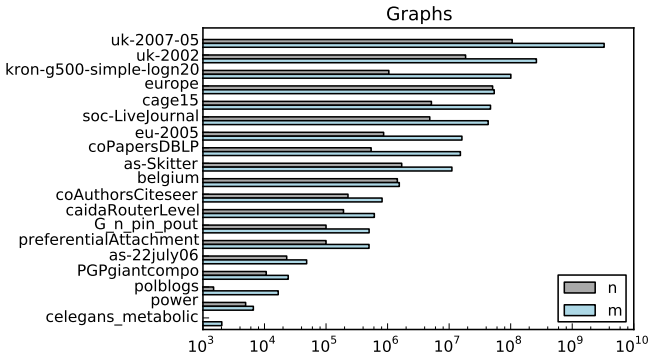

Figure 4. Size comparison of all graphs used in experiments

when all nodes have the label of the relative majority in their neighborhood.

Label propagation can be parallelized easily by dividing the range of nodes among multiple threads which operate on a common label array. This parallelization is not free of race conditions, since by the time the neighborhood of a node u is evaluated in iteration i to set $\zeta_i(u)$, a neighbor v might still have label $\zeta_{i-1}(v)$ or already $\zeta_i(v)$. The clustering outcome thus depends on the order of threads. However, these race conditions are acceptable and even beneficial in an ensemble setting since they introduce random variations and increase base clustering diversity. This also corresponds to *asynchronous updating*, which has been found to avoid oscillation of labels on bipartite structures [15].

When dealing with scale-free networks whose degree distribution follows a power law, assigning node ranges of equal size to each thread can lead to load imbalance as computational cost depends on the node degree. Instead of statically dividing the iteration among the threads, guided scheduling (with `parallel for schedule(guided)`) can help to overcome load balancing issues, although this introduces some overhead. We observed that dynamic scheduling is generally superior to static parallelization in terms of PLP's speed.

	category	real-world data?	set
uk-2007-05	web	yes	L
uk-2002	web	yes	TL
kron-g500-simple-logn20	Kronecker	no	TL
europe	street	yes	L
cage15	biophysics	yes	TL
soc-LiveJournal (undir.)	social	yes	L
eu-2005	web	yes	L
coPapersDBLP	coauthorship	yes	TL
as-Skitter	internet	yes	L
belgium	street	yes	T
coAuthorsCiteseer	coauthorship	yes	T
caidaRouterLevel	internet	yes	T
G_n_pin_pout	random	no	T
preferentialAttachment	random	no	T
as-22july06	internet	yes	T
PGPgiantcompo	social	yes	T
polblogs	web	yes	T
power	powergrid	yes	T
celegans_metabolic	biochemistry	yes	T

Table I
GRAPHS USED IN EXPERIMENTS

B. Parallel Louvain Method

Our implementation of PLM (Algorithm 2) employs parallel iteration over the node set. Since the computation of the Δmod scores is the most frequent operation, it needs to be very fast. That is why we store and update interim values, which is not apparent from the high-level pseudocode in Algorithm 2. To this end, we associate with each node a map in which the Δmod values for neighboring clusters are stored and updated when node moves occur. We can thus avoid the costly recomputation for each possible node move. A lock for each vertex v protects all read and write accesses to v's map since `std::map` is not thread-safe. Two strategies are available for apportioning the node set among the threads, simple (static scheduling) and balanced (guided scheduling), whose effects are discussed in Section VII-B.

C. Ensemble Techniques

Our implementations of the ensemble techniques EML and EPP are agnostic to the base and final algorithms and can be instantiated with a variery of clusterers. We implemented EML so that it stops if the modularity value has not been improved for three levels. As argued in Section IV-C3, the use of a b-way hash function is advantageous for creating the core clustering, as it is significantly faster than region growing due to a high degree of parallelism. We use a relatively simple function called djb2 due to Bernstein,[3] which appears sufficient for our purposes.

VI. Experimental Setup

A. Graphs

We perform experiments on a variety of graphs from different categories of both real-world and synthetic data sets—among them web graphs, internet topology networks, social networks, scientific coauthorship networks and street networks (see Table I). Therefore, we cover a wide range of graph-structural properties, from regular meshes to complex networks which show a non-trivial combination of randomness and regularity. Note that the achievable modularity depends on the structural characteristics of the graph, such as an inherent community structure, which may or may not be distinctive.

The majority of our test graphs are taken from the collection compiled for the *10th DIMACS Implementation Challenge* and are freely available on the web.[4] They are undirected, unweighted graphs stored in files of the METIS adjacency format. Two additonal large complex networks, as-Skitter and soc-LiveJournal, are derived from the *Stanford Large Network Dataset Collection*.[5] We create an undirected version of soc-LiveJournal, for which we replace each directed edge by an undirected edge and delete multiple edges.

In order to evaluate algorithm engineering decisions, we use a fixed subset of the DIMACS graphs (set T), selected for its broad range of network categories and sizes. To demonstrate the scalability of our algorithms, experiments are performed with an additional set of large graphs (set L), which focuses on complex networks. For experiments on Platform 2, we also add the largest available graph uk-2007-05, a web graph of the .uk domain with $n \approx 105 \cdot 10^6$ and $m \approx 3.3 \cdot 10^9$, which needs more than 250 GB of memory in the course of an EPP run. Table I lists the complete set of graphs and gives a short characterization of each.

B. Settings

For representative experiments, we average quality and speed values over 5 runs in order to compensate for fluctuations. Table II provides information on the two platforms used. The memory capacity of Platform 2 is needed to handle the largest available graph uk-2007-05, but it is generally slower than Platform 1, which we used to obtain the best results. All running times are given in seconds.

	Platform 1 (compute11.iti.kit.edu)	Platform 2 (ic2.scc.kit.edu)
compiler	gcc 4.7.1	gcc 4.7.2
OS	SUSE 12.2-64	SUSE ES 11 SP2
CPU	2x 8-Core Xeon E5-2670, 2.6 GHz	4x 8-Core Xeon E7-8837, 2.67 GHz
RAM	64 GB	512 GB

Table II
Platforms used in experiments

[3]hash functions: http://www.cse.yorku.ca/~oz/hash.html
[4]*DIMACS* collection: http://www.cc.gatech.edu/dimacs10/downloads.shtml
[5]*Stanford* collection: http://snap.stanford.edu/data/index.html

VII. Experiments and Results

In this section we report on a representative subset of our experimental results for our different parallel algorithms.

A. Parallel Label Propagation

PLP is extremely fast and able to handle the large graphs easily (Figure 6). The 3.3 billion edge web graph uk-2007-05 is processed in under 120 seconds with 32 threads working in parallel on Platform 2. The "weak classifier" PLP is nonetheless able to detect an inherent community structure and produce a clustering with reasonable modularity values, although it cannot distinguish communities in a Kronecker graph, which has a very weak community structure.

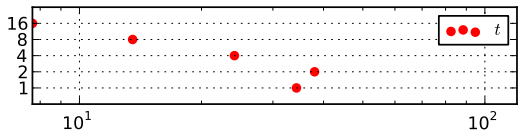

Figure 5. Strong scaling of PLP on the uk-2002 web graph (Platform 1, running time in s)

For weak scaling experiments, we use a series of synthetic Kronecker graphs where each graph has twice the size of its predecessor (from $\log n = 16 \ldots 21$), and double the number of threads simultaneously from 1 to 16. PLP exhibits good weak scaling, apart from some overhead introduced by *OpenMP* parallelization. To demonstrate strong scaling behavior, we apply PLP to the large uk-2002 web graph and increase the number of threads from 1 to 16 (Figure 5). We observe linear strong scaling in the range of 2-16 threads. We conjecture that the exception when transitioning from 1 to 2 threads is caused not only by *OpenMP* overhead, but also by the *Intel SpeedStep* technology of Platform 1, where processor clock frequency is adapted to the number of cores currently utilized.

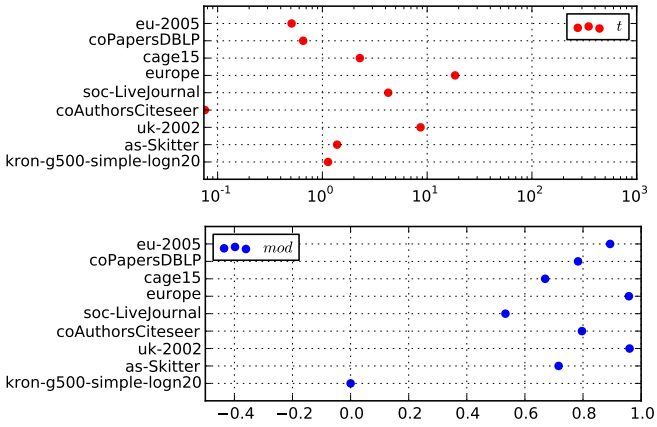

Figure 6. PLP handles large graphs in seconds and detects meaningful community structure on most instances (Platform 1, running time in s)

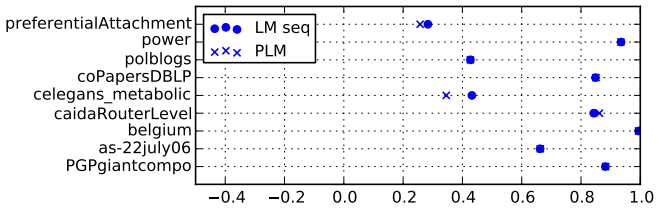

Figure 7. Clustering quality for PLM and its sequential variant

B. Parallel Louvain Method

As Figure 7 shows, we observe only small deviations in clustering quality between our sequential and parallel implementation, supporting the argument that PLM is able to correct undesirable decisions due to stale data. Note that PLM finds higher quality clusterings than PLP but is too slow for our purposes on large graphs. Comparing simple and balanced parallelization on graph set T (Figure 9), we observe that quality and speed vary depending on the graph. Using guided scheduling may result in both speedup and slowdown, as well as loss or gain of modularity. We conjecture that the order in which nodes are visited has an influence on clustering quality. The observation that some graph instances are affected by the choice of strategy while others are not might be explained by differences in sort order in the graph input file: If e.g. the high-degree nodes present in complex networks are accumulated in the beginning of the index range, the order in which nodes are considered can lead to significant differences in clustering outcome. Therefore, we confirm that the choice between simple and balanced parallelization should remain a configurable parameter of the implementation. In the following, we use the balanced variant as default.

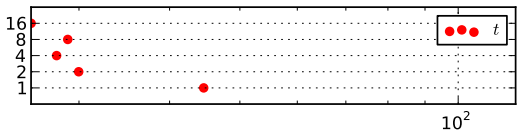

Figure 8. Strong scaling of PLM on the eu-2005 web graph (Platform 1, running time in s)

PLM shows worse weak scaling than PLP, with about 50% increase in running time per doubling step. We use the smaller web graph eu-2005 to demonstrate strong scaling behavior of PLM. Evidently, the algorithm can benefit from increased parallelism, but less predictably so compared to PLP. Overhead is introduced by the locking of data structures which PLM requires in contrast to PLP. In addition to this, parallelism can lead the algorithm to take different control paths and possibly terminate earlier, leading to irregular scaling results.

C. Ensemble Techniques

1) Ensemble Multilevel: Our experiments with EML show that the iterated scheme does not pay off in terms of quality

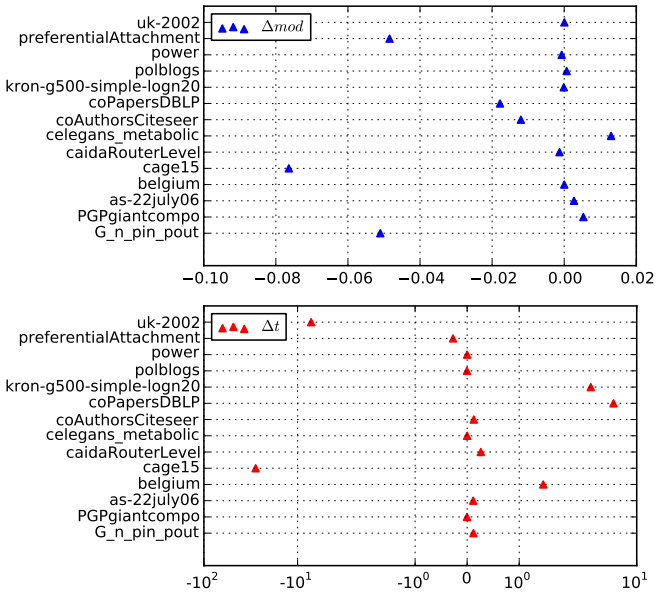

Figure 9. Differences in quality (above) and running time (below) when switching from simple to balanced parallelism in PLM (Platform 1, running time in s)

in most cases. Usually the quality obtained on the first or second level cannot be further improved. This contrasts results by Ovelgönne and Geyer-Schulz [14]. One reason for this discrepancy may well be small differences in our implementation of label propagation. Furthermore, it seems that PLP is not the ideal sole base clusterer on coarser levels. There its clusterings become quite similar. Similar base clusterings limit the core clustering optimization process. EPP, discussed next, is therefore faster and even improves quality compared to EML.

2) Ensemble Preprocessing: Figure 10 shows an EPP instance with a 4-piece PLP ensemble and PLM as final clusterer in comparison to a single PLP instance. We observe that the approach of EPP pays off in the form of improved modularity values on all but one instance. In the latter case, PLP cannot detect the extremely weak community structure in the first place. This comes at the cost of a running time about 10 times higher. Still, large graphs are easily handled in a few seconds to minutes. On Platform 2, a modularity of 0.99598 is reached for the uk-2007-05 graph in 660 seconds.

We vary the ensemble size, doubling the number of base clusterers from 1 to 8, and observe the difference in modularity (Figure 11). On average, clustering quality can be gained by increasing the ensemble size, although the actual difference depends strongly on the individual graph. On a few small graphs, a bigger ensemble makes the difference between failure to distinguish communities and a clustering with reasonable quality, while on a large graph like uk-2002, the difference is near zero. A loss in quality is rare but was observed for the coPapersDBLP network. Running time increases at least proportionally to the number of base clusterers. We therefore

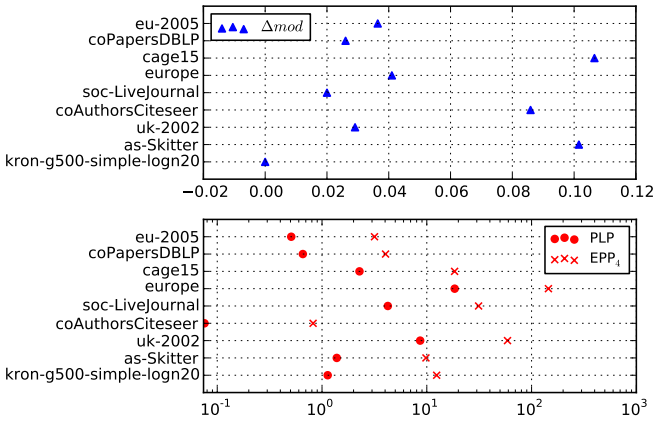

Figure 10. Quality improvement (above) and absolute running time (below) of EPP_4 compared to a single PLP (Platform 1, running time in s)

conclude that forming a large ensemble is not justified, since a small ensemble already provides quality improvement, and settle on four base clusterers for the remaining experiments.

Combining the results of multiple classifiers in an ensemble learning scenario is only meaningful if the base classifiers disagree about some elements. Therefore, we inspected the diversity of the base clusterings by calculating the *Jaccard* dissimilarity measure [10], observing that the clusterings produced my multiple PLP instances are not necessarily different, that the dissimilarity varies non-deterministically between multiple runs, and that quality produced by EPP depends on the degree of dissimilarity to some extent. As a possible solution for creating diversity among the clusterings, explicitly randomizing the order in which PLP traverses the nodes becomes interesting again in an ensemble setting. However, we find that explicit randomization has no significant effect on clustering quality in an ensemble setting, while it slows down the algorithm for large graphs. We therefore confirm that explicit randomization should be omitted. As an alternative solution, we try to perturb the clustering initially by randomly choosing a small number of seed nodes and deactivating them, or activating only this seed set. However, deactivation of seed nodes does not seem to influence clustering diversity or result quality in a reproducible way.

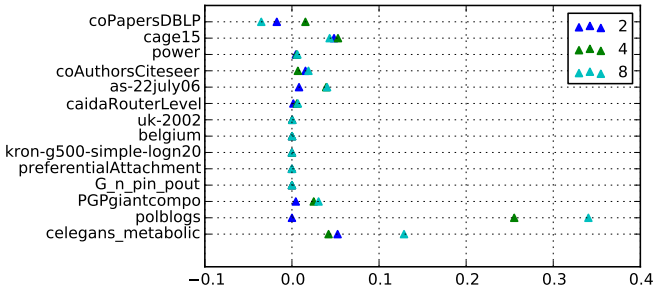

Figure 11. Modularity differences when increasing the ensemble size from 1 to 2, 4 and 8 base clusterers

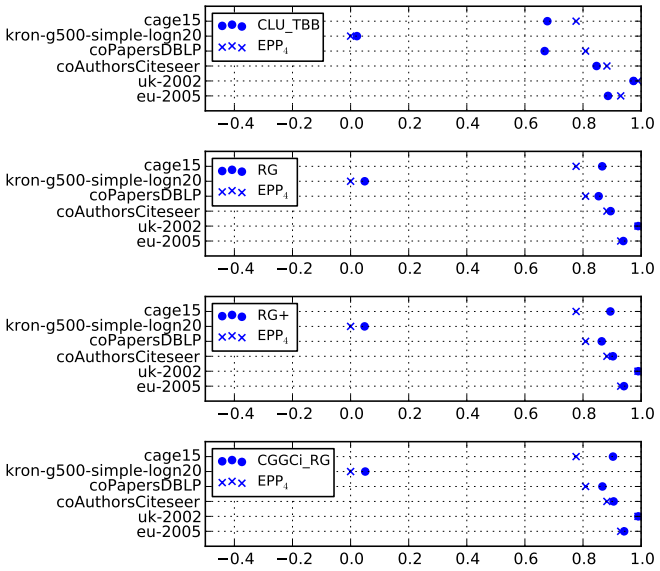

Figure 12. Quality comparison of EPP to *DIMACS* challenge competitors

D. *Comparison with* DIMACS *competitors*

In the following, we perform a comparision of our best performing algorithms to clusterers which excelled in the *DIMACS* challenge either by solution quality or time to solution. With modularity, we have a widely accepted objective measure of clustering quality, but there is no accepted normalization for running times on different machines. Therefore, we would like to stress that running times were measured on different platforms and comparability is limited. That said, the comparison does provide some insight on how our implementation performs with respect to state-of-the-art competitors.

Figure 12 shows modularity values for EPP_4 in comparison with CLU_TBB, RG, RG+ and CGGCi_RG on a set of large graphs which is the intersection of our set L and the graphs in the final challenge testbed. Within the limits of comparability, EPP_4 and CLU_TBB, the fastest competitor, show running times of the same magnitude (60 and 30 seconds on the uk-2002 graph). In turn, EPP achieves significantly better modularity values on some graphs. Overall, RG and RG+ clearly achieve better quality on some graphs, but are decidedly slower. EPP is one magnitude faster than RG and more than two orders of magnitude faster than Pareto winner RG+, which implements an ensemble technique very similar to that of EPP. CGGCi_RG, the multilevel ensemble scheme using the RG algorithm, is also superior in terms of quality, but nearly 4 orders of magnitude slower (with running times in the range of 100 hours). We conclude that EPP is not dominated by any other clusterer to which we were able to compare it. Moreover, if the quality of PLP is sufficient for an application, this algorithm should be considered since it is faster than all other competitors.

VIII. Conclusion and Future Work

For this paper we have developed and implemented parallel algorithms for community detection, a common clustering task in network analysis. Successful combinations and parameter settings have been identified in extensive experiments on synthetic and real-world networks. While the PLP algorithm is extremely fast, its quality might not always be satisfactory for some applications. PLM is to the best of our knowledge the first parallel variant of the established Louvain algorithm for massive inputs. Its combination with an ensemble of PLP base clusterers yields the strong clusterer EPP that can cluster a graph with 3.3 billion edges in 11 minutes. Thus it delivers the best tradeoff between quality and speed compared to the state of the art.

Moreover, we have introduced an algorithmic framework whose extensibility and flexibility allow a seamless addition of further high-performance network analysis functionality. We invite other researchers to contribute to this effort.

We have exhausted the *DIMACS Challenge* and *Stanford* graph collections in terms of graph size without reaching the limits of our algorithms and available hardware in terms of running time. Therefore, future work is concerned with efficient practical algorithms for other, possibly more complex, network analysis and optimization tasks, and also for dynamic graphs from data streams. In practice, the memory footprint of large graphs is more relevant as a limiting factor than the execution time of the PLP and EPP algorithms, but the compactness of our graph data structure can be improved.

Also, on some of our small graphs, the PLP algorithm occasionally blunders, e. g., it merges two natural communities, which significantly degrades the modularity scores. A deeper analysis of this behavior is necessary, but difficult due to non-determinism. Additional techniques might help, such as counteracting "epidemic spread" of labels as examined in [19].

Acknowledgements: This work was partially supported by the project *Parallel Analysis of Dynamic Networks — Algorithm Engineering of Efficient Combinatorial and Numerical Methods*, which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg.

References

[1] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[2] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[3] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Trans. Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[4] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[5] B. O. Fagginger Auer and R. H. Bisseling, "Graph coarsening and clustering on the GPU," in *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[6] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75 – 174, 2010.

[7] S. Fortunato and M. Barthelemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[8] J. Gilbert, S. Reinhardt, and V. Shah, "High-performance graph algorithms from parallel sparse matrices," Mar. 2006.

[9] M. Girvan and M. Newman, "Community structure in social and biological networks," *Proc. of the National Academy of Sciences*, vol. 99, no. 12, p. 7821, 2002.

[10] P. Jaccard, *Distribution de la Flore Alpine: dans le Bassin des dranses et dans quelques régions voisines*. Rouge, 1901.

[11] K. Kothapalli, S. Pemmaraju, and V. Sardeshmukh, "On the analysis of a label propagation algorithm for community detection," in *Distributed Computing and Networking*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7730, pp. 255–269.

[12] M. Müller-Hannemann and S. Schirra, Eds., *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*, ser. Lecture Notes in Computer Science, vol. 5971. Springer, 2010.

[13] M. Ovelgönne, "Distributed community detection in web-scale networks," University of Maryland, Tech. Rep., July 2012.

[14] M. Ovelgönne and A. Geyer-Schulz, "An ensemble learning strategy for graph clustering," in *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[15] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.

[16] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[17] R. Rotta and A. Noack, "Multilevel local search algorithms for modularity clustering," *J. Exp. Algorithmics*, vol. 16, pp. 2.3:2.1–2.3:2.27, Jul. 2011.

[18] S. E. Schaeffer, "Graph clustering," *Computer Science Review*, vol. 1, no. 1, pp. 27–64, 2007.

[19] J. Soman and A. Narang, "Fast community detection algorithm with gpus and multicore architectures," in *Proc. 25th IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 568–579.

[20] C. Staudt, A. Schumm, H. Meyerhenke, R. Gorke, and D. Wagner, "Static and dynamic aspects of scientific collaboration networks," in *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*. IEEE, 2012, pp. 522–526.

[21] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. ACM, 2012, p. 3.

[22] Y. Zhang, J. Wang, Y. Wang, and L. Zhou, "Parallel community detection on large networks with propinquity dynamics," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 997–1006.